SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Improving Responsiveness of Time-Sensitive Applications by Exploiting Dynamic Task Dependencies

Tommaso Cucinotta[1]* Luca Abeni[1], Juri Lelli[2], Giuseppe Lipari[3]

[1]*ReTiS Lab, Scuola Superiore S.Anna, Via G. Moruzzi, 1 - 56124 Pisa, Italy*
[2]*ARM Ltd., Cambridge (UK)*
[3]*Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France*

SUMMARY

In this paper, a mechanism is presented for reducing priority inversion in multi-programmed computing systems. Contrarily to well-known approaches from the literature, this paper tackles cases where the dependency relationships among tasks cannot be known in advance to the operating system (OS). The presented mechanism allows tasks to explicitly declare said relationships, enabling the OS scheduler to take advantage of such information and trigger priority inheritance, resulting in reduced priority inversion. We present the prototype implementation of the concept within the Linux kernel, in the form of modifications to the standard POSIX condition variables code, along with an extensive evaluation including a quantitative assessment of the benefits for applications making use of the technique, as well as comprehensive overhead measurements. Also, we present an associated technique for theoretical schedulability analysis of a system using the new mechanism, which is useful to determine whether all tasks can meet their deadlines or not, in the specific scenario of tasks interacting only through remote procedure calls, and under partitioned scheduling. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Priority Inversion; Priority Inheritance; Real-Time Scheduling on Linux; Concurrent Programming; Real-Time Analysis

1. INTRODUCTION

A broad class of computing systems, from traditional embedded/cyber-physical systems and personal computing to cloud and distributed infrastructures, are challenged nowadays by the increasing need for hosting time-sensitive and interactive workloads with precise timing and Quality of Service (QoS) constraints. These pose unprecedented demands on the underlying resource management and scheduling mechanisms in terms of responsiveness and flexibility. A particularly critical resource to manage in this context is the processor. Indeed, the way CPUs are allocated in a distributed environment, and the way they are temporally scheduled by the underlying operating system (OS) and kernel, constitute the foundation on top of which interactive services meeting tight response-time constraints can be built.

In this context, proper scheduling and prioritization of software components becomes key to ensure low latency, dynamism and responsiveness of applications and services under highly variable workload conditions. However, even if all the applications in the system are assigned appropriate priorities, and if the CPU scheduler makes its best to respect such priorities, interaction between the

*Correspondence to: ReTiS Lab, Scuola Superiore S.Anna, Via G. Moruzzi, 1 - 56124 Pisa, Italy. E-mail: tommaso.cucinotta@santannapisa.it

various threads or processes can affect the response times. This can happen, for example, because of the so called *priority inversion* problem, occurring whenever the execution of an important task is delayed due to the interference from other less important tasks. This typically happens when the important task is blocked waiting for another task, which is preempted by a less important task. Albeit such problem has been largely investigated in the real-time systems literature (see Section 7), there are still many recurrent situations in which priority inversion could be avoided if the operating system provided the right mechanisms and abstractions to enable applications to communicate more about their concurrency structure and tasks' inter-dependencies. This is witnessed for example by recent works aimed at improving the real-time responsiveness of the Android OS [1, 2], specifically on the side of OS services for inter-process communication.

1.1. Contributions

In this paper, a novel mechanism for reducing priority inversion in multi-programmed systems is presented. It allows the OS kernel to take advantage of the explicit declaration of the dependencies between tasks. Whenever a task waits for another task (the "*helper*") to complete some action, the OS kernel dynamically boosts the priority of the helper, applying a form of *priority inheritance*. The mechanism has been implemented in a Linux-based OS around the *condition variables* programming abstraction (defined by POSIX [3]), through proper modifications to the futex code within the Linux kernel, and a tiny addition to the userspace libraries. Overhead figures from the real implementation are shown, proving feasibility of the technique.

Notice that, although the presented implementation is based on condition variables as available through the `pthreads` library, the idea is generic and is not strictly bound to condition variables (which are just an implementation detail). Also note that while the priority inheritance mechanism has been traditionally applied to resource sharing (competition synchronization) or blocking remote procedure call (RPC) mechanisms, this paper shows how to apply it to generic synchronization operations, including cooperation synchronization.

The conceptual design of the new mechanism has been preliminarily presented in [4], where the technique had been prototyped within an open-source simulator for real-time scheduling, and results were gathered in simulation and from a simple RPC scenario only. In the present paper, we complete that preliminary work by presenting for the first time:

1. a thorough validation of the concept based on a real implementation within the Linux OS, in the form of modifications to the futex code-base in the kernel, and to the standard condition variables `pthreads` API for user-space applications;
2. a comprehensive evaluation of the overheads of the technique;
3. an assessment of the benefits of the technique based on measurements done on a synthetic application scenario;
4. an example of theoretical schedulability analysis of real-time applications making use of the new proposed mechanism (limited to the case of synchronous RPC interactions and partitioned scheduling, with each task interacting only with other tasks on the same CPU), showing that the proposed technique can also improve the schedulability in hard real-time systems.

1.2. Paper organization

This paper is organized as follows. In Section 2, we provide background information on real-time scheduling and motivate the need for a technique like the one presented in this work. In Section 3, we propose priority inheritance on Condition Variables (PI-CV) as a means for reducing priority inversion in time-sensitive applications. In Section 4, we describe a realization of the proposed technique within the Linux kernel, accompanied by a proper user-space API that integrates nicely with the `pthreads` library. In Section 5, we provide a real-time analysis methodology for schedulability analysis of a real-time system enriched with the proposed mechanism. In Section 6, we first show what benefits are brought in by PI-CV in terms of tasks performance (Section 6.2), then we provide overhead measurements demonstrating viability of the technique in various workload

DOI: 10.1002/spe

conditions (Section 6.3). Finally, in Section 7, we briefly recall related work in the research literature, just before drawing conclusions in Section 8, where we also discuss possible future work on the topic.

2. BACKGROUND

In multi-programmed computing systems, there is often the need to concurrently run multiple activities with different importance and urgency. For example, in real-time and embedded systems, it is commonplace to deploy activities with different importance and/or criticalness at different priority levels. In operating systems design, drivers carrying out I/O interactions with external peripherals need to execute as soon as possible, to guarantee prompt reactivity of the computing system with respect to external events, and sometimes actual execution of the drivers code is handed over to user-space daemons, as it commonly happens in Linux PREEMPT_RT. In virtualized computing infrastructures supporting virtualized network functions and cloud computing, it is often the case that a Virtual Machine Monitor (VMM) creates multiple threads for emulating the behavior of a single Virtual Machine (VM). For example, KVM[†] runs as a process in Linux, creating a thread for each emulated Virtual CPU of the VM, plus additional threads (depending on the chosen para-virtualization options) for handling the virtualized I/O of the VM, including networking. In such a case, the I/O threads have to run before the threads emulating virtual cores, lest a poor performance of the VM from a networking latency and throughput perspective, particularly in scenarios with CPU-intensive workloads running within the VM.

In all these cases, the CPU(s) available on the computing system are dynamically assigned to the different activities by the OS or VMM scheduler, based on priority levels that are assigned to them. Such a fixed priority (FP) scheduler, allowing activities having higher priorities to run before others with lower priorities, is very recurrent in computing systems designs. On the other hand, the reader should assume that whenever this document refers to priorities of tasks, it actually refers more generally to their urgency of execution from the CPU scheduler viewpoint, this being represented as a classical integer priority level, a deadline as with schedulers based on Earliest Deadline First (EDF), or other types of time-stamps, such as virtual start-time as used in fair schedulers (such as the Linux CFS), or others.

In complex software infrastructures, it often happens that these different components running at possibly different priority levels may need to interact with each other, e.g., for accessing common middleware or OS services, such as messaging, monitoring, logging, etc., leading to potential *priority inversion* scenarios. This problem has been previously considered in literature and industrial practice by considering interactions involving blocking primitives on a mutual exclusion semaphore (*mutex* in the following): a higher-priority task is waiting to acquire a mutex lock held by a lower-priority task, which is preempted by a middle-priority task that ultimately causes delays to the higher-priority task that is waiting to enter the critical section. This problem can be tackled within the OS kernel with the classical *priority inheritance* protocol (see Section 7): a task inherits dynamically the highest among the priorities of all tasks waiting for mutexes it has locked, if said highest priority is higher than its own one. Priority inheritance is widely available within nowadays operating systems, e.g., through the real-time extensions of the POSIX standard [3], supported for example on Linux via the `pthreads` API (setting the `PTHREAD_PRIO_INHERIT` protocol on the mutex) and corresponding to what is called a *rt-mutex* at the kernel level.

The priority inheritance mechanism has been designed around competition synchronization (synchronization over mutual exclusion semaphores). Whenever some other kind of synchronization is required (for example, cooperation synchronization [5], when a task blocks waiting for some data coming from other tasks, or waiting for other tasks to finish some kind of action), the traditional priority inheritance mechanism does not help, due to the lack of knowledge of the dependency relationships among tasks. This paper deals with equally common and widely recurrent

[†]More information is available at: `http://www.linux-kvm.org/page/Main_Page`.

DOI: 10.1002/spe

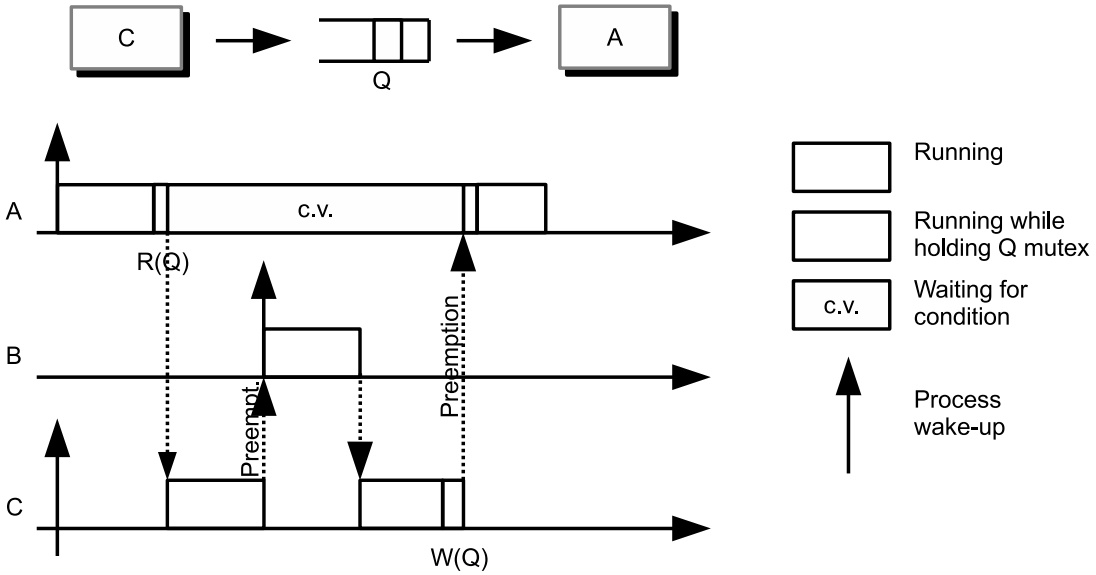

Figure 1. Priority inversion scenario with task A receiving data from a lower-priority task C through a shared message-queue Q. Task B has middle-priority between A and C.

scenarios where priority inversion still happens, because it cannot be addressed by traditional priority inheritance on competition synchronization.

Consider for example the scenario depicted in Figure 1, where we have three tasks, A, B and C, with decreasing priority order. Task A waits for some data from C (passed through a message queue Q), calling a blocking receive operation on Q. Unfortunately, while C is generating the data for A, a middle-priority task B wakes up, causing delays in the execution of C, and ultimately delaying A. The traditional priority inheritance mechanism cannot help here, because it is designed around critical sections, in which the OS has *knowledge about the dependency* among tasks, namely which task is currently in the critical section. However, Task C is not holding any mutex lock while progressing towards completing the computations that will lead to the production of a data item to be pushed into the message queue Q. In addition, the system has no prior knowledge of the fact that it will be C that will generate the data A is waiting for. This is an example of priority inversion scenario that can be addressed by using the technique proposed in this paper (see Section 3): letting the OS know that the high-priority task A is blocked (because of cooperation syncrhonization, not because of competition for a shared resource) waiting for some output *to be provided by task C*, allows the OS kernel to forbid the middle priority task B to preempt task C in such a situation. See Section 6 for an example of how such message queue might be implemented using, for example, POSIX threads and *condition variables*, along with examples of real execution of a synthetic producer/consumer RPC scenario using it.

Another classical example of priority inversion that can be reduced/controlled by using our proposed technique is the one of a client-server interaction with a software component (the server) that is part of some middleware, or embedded within the OS, that may perform some system-level action on behalf of the caller software component (the client). In presence of clients with multiple different priorities, if such a server is assigned a high priority level, then it might prevent a high priority client to run even while it is serving a client with a low priority, causing a form of priority inversion. On the other hand, if the server is given a low priority, then, even while serving a request on behalf of a high priority client, it might be preempted and delayed by a middle priority client, causing again priority inversion.

Techniques for addressing these further priority inversion issues include priority inheritance techniques, as investigated in the Ada language by Sha [6] back in 1987-1990, or more recently for client-server interactions in reservation-based systems by Abeni et al. [7].

DOI: 10.1002/spe

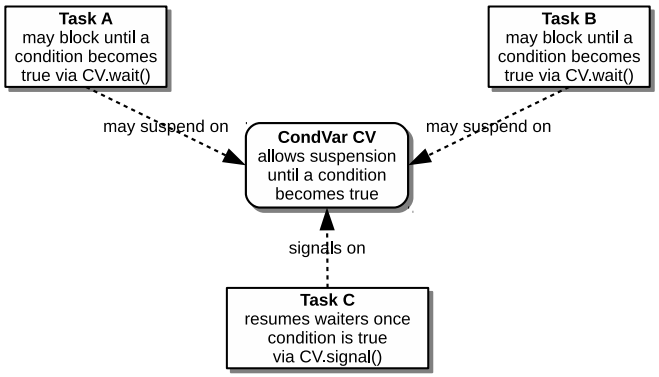

Figure 2. Two tasks A and B potentially suspending through a `wait()` on condition variable CV, where Task C will perform a `signal()` to wake-up waiters.

3. PRIORITY INHERITANCE ON COOPERATION SYNCHRONIZATION

As explained above, this paper addresses the problem of avoiding priority inversion in scenarios with multiple interacting tasks running at different priorities in the OS, with focus on *blocking interactions*. While in some situations the dependency relationship between these tasks is implicitly known (for example, consider synchronous client-server calls or RPCs), in this paper we address more generic interactions where such dependency cannot be known in advance. For example, consider generic interactions happening via *condition variables* (see for example Figure 2): these allow one or more threads (e.g., tasks A and B in the figure) to suspend, via a `wait()` operation, until another thread (e.g., task C in the figure) performs a `signal()` operation, but the run-time has no prior knowledge of which thread will do that. Similarly, when communicating through a named FIFO on the file-system, a task can block waiting for other tasks to provide data by writing to the same FIFO, but the run-time lacks knowledge about what other task(s) in the system might actually perform the write.

In what follows, without loss of generality, the term *task* will be used to refer to a single thread of execution in a computing system. Also, without loss of generality, the term *priority* will be used to refer to the right of execution (or "urgency" level) of a task as compared to other tasks from the CPU(s) scheduler viewpoint. This includes the priority of tasks whenever they are scheduled according to a priority-based discipline, their deadline whenever they are scheduled according to a deadline-based discipline, and their time-stamp whenever they are scheduled according to other policies based for example on virtual times, such as the Linux Completely Fair Scheduler (CFS) [8]. However, the described technique is not specifically tied to any of these scheduling disciplines and it can be applied in presence of other schedulers as well. Furthermore, it should be clarified that this paper deals with how to let tasks dynamically inherit priorities (or right of execution) among one another, which is orthogonal w.r.t. which scheduling algorithm is being used.

In this paper, we propose PI-CV, a mechanism triggering priority inheritance whenever a task suspends on a blocking interaction via condition variables (CVs). PI-CV constitutes an extension to the condition variables interface and kernel-level implementation: it allows explicit declaration of the set of tasks, called *helpers*, that are supposed to "help out" in the realization of the general condition that a task blocked on a condition variable is waiting for. Said set can be fixed throughout the CV life-time, or be dynamically changed at run-time, according to the application needs. For example, in a classical producer-consumer blocking interaction scenario, a producer pushes items into a shared queue, while the consumer pops items out of it. When the queue is empty, the consumer blocks waiting for additional items to be pushed. In such a setting, it is normally statically known which task is the producer and which one is the consumer, so the producer task can be added to the set of helpers for the condition variable used by consumers to block on an empty queue.

With PI-CV, whenever a higher-priority task executes a `wait()` operation on a CV having a non-null set of helper tasks, it temporarily donates its priority to all the lower-priority helper tasks, so as

DOI: 10.1002/spe

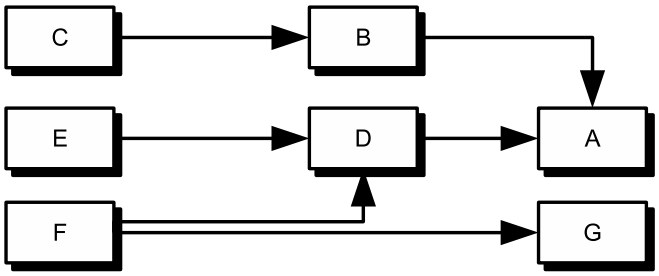

Figure 3. General interaction scenario where priority inheritance on condition variables may be applied transitively. Task F is waiting on a condition variable having tasks D and G registered as helpers.

to "speed-up" their progress towards performing the corresponding `signal()` operation. At this time, the dynamically inherited priority is revoked, restoring the original priority of the helper tasks. PI-CV can be nicely integrated with traditional priority inheritance on mutexes and semaphores, resulting in priority being inherited from a higher priority task to a lower priority one either because the former waits to acquire a lock held by the latter, or because the former is suspended due to a wait operation on a CV for which the latter is a helper task.

In order for the mechanism to work, it is necessary to introduce a few interface modifications to the classical CV mechanism, so that the operating system knows which lower-priority tasks should inherit the priority of a higher-priority task suspending its execution waiting for a condition to become true. The interface allows the mechanism of priority inheritance on CVs to be enabled selectively on a per-CV basis, depending on the application requirements.

Priority inheritance may be applied *transitively*, when needed. For example, if Task A blocks on a CV donating temporarily its priority to Task B, and Task B in turn blocks on another condition variable donating temporarily its priority to Task C, then Task C should inherit the highest priority among the one associated with all the 3 tasks. Also, PI-CV can be integrated with traditional priority inheritance (or deadline inheritance) as available on current operating systems, letting the priority transitively propagate either due to an attempt of locking a locked mutex, or to a suspension on a CV with associated one or more helper tasks.

Consider a blocking chain of tasks $(\tau_1, \tau_2, \ldots, \tau_n)$ where each task τ_i $(1 \leq i \leq n-1)$ is suspended on the next one τ_{i+1} either trying to acquire a lock (enhanced with priority or deadline inheritance) already held by τ_{i+1}, or waiting on a condition variable (enhanced with PI-CV as described in this document) where τ_{i+1} is registered among the helper tasks. All the tasks in such a blocking chain are suspended, except the last one (that is eligible to run). This last task inherits the highest priority among the tasks in any blocking chain terminating on it, i.e., any task in the direct acyclic graph (DAG) of blocking chains that terminate on it.

For example, consider the scenario shown in Figure 3, where each arrow from a task to another means that the former is suspended on the latter due to either a blocking lock operation or a wait on a CV where the latter task is one of the helpers. Task A inherits the highest priority among tasks B, C, D, E, F (if higher than A's own one), while G inherits the priority of F (if higher than G's own one), if all of the suspensions happen through mutex semaphores enriched with priority inheritance or CV enriched with PI-CV. Note that F is waiting on a CV where both D and G are registered as helpers. This allows both of them to inherit the priority of F, until the condition is notified.

4. IMPLEMENTATION

In this section, we describe the user-space API calls that we designed to support PI-CV in a way that is as POSIX-oriented as possible, namely flanking it to the `pthreads` library. Then, we provide details on how the mechanism has been realized in the Linux kernel.

DOI: 10.1002/spe

4.1. User-space interface

From an interface standpoint, the proposed mechanism is made available to applications via a specialized library call that can be used by a task to declare which other tasks are the potential helpers towards the verification of the condition associated with a condition variable. In our implementation, based on the `pthreads` library, this is realized through the following C library calls:

```
int pthread_cond_helpers_add(pthread_cond_t *cond, pid_t helper);
int pthread_cond_helpers_del(pthread_cond_t *cond, pid_t helper);
```

These two functions add or delete the `helper` thread to the pool of threads (empty after a `pthread_cond_init()` call) that can potentially inherit the priority of any thread waiting on the condition variable `cond` by means of a `pthread_cond_wait()` or `pthread_cond_timedwait()` call.

For convenience, our introduced new `pthreads` calls are currently made available through a separate library, and they are realized directly in terms of low level syscalls (i.e., `sys_futex()`, see below). The `pthreads` library uses `pthread_t` values to identify individual threads, however in our current implementation we pass the Linux thread ID of helpers (as available through `gettid()` syscall). This can easily be changed in the future, though.

In our kernel level implementation, the condition variable is associated with a list of helper threads, and a kernel-level modification ensures that the highest priority among the ones of all the waiters blocked on the condition variable is dynamically inherited by the registered helper thread(s), whenever higher than their own priority (and also that this inheritance is transitively propagated across both condition variables and rt-mutexes supporting priority inheritance). Whenever the `pthread_cond_signal()` or `pthread_cond_broadcast()` function is called, the corresponding woken-up thread(s) will revoke donation of their own priority.

In what follows, we describe the kernel-level changes that were necessary to integrate PI-CV with traditional priority inheritance in the Linux futex code base. We refer the reader to [9] for the needed background information.

4.2. Data structures

Major changes to in-kernel data structures are summarized in Figure 4, where the added fields and data structures are highlighted in *italics*. In the following, for the sake of brevity, we report just elements that are essential to understand what changes we made to the in-kernel data structures, simplifying the explanation of the code in various aspects, e.g., omitting aspects related to the synchronization for concurrent access.

Futexes are looked up in the kernel through the global `__futex_data` structure, using a `futex_key` data structure as look-up key. This is roughly equivalent to hashing the user-space virtual address of a process-private futex, or its physical address, for process-shared ones. The `__futex_data.queues` field is a hash-table where each `futex_hash_bucket` points (via the `chain` field) to a priority list (`plist.h`) of `futex_q` nodes. Each such node corresponds to a task (pointed to by `futex_q.task`) blocked on a `pthread_cond_wait()` operation on the futex whose exact key is contained in the `futex_q.key` field. On a `pthread_cond_signal()`, in order to find the top-priority waiter task to be woken up for a futex with key hashing to `h`, the kernel scans the `__futex_data.queues[h]->chain` priority list of `futex_q` nodes till it finds the first node with the matching correct key. On a `pthread_cond_broadcast()` instead, all waiters to be woken up can be found by scanning the same list till the end, and retrieving the tasks from `futex_q` nodes with the right key.

We added a second hash-table `__futex_data.helpers`, where the futex key is used to look-up a `futex_hash_bucket` whose `chain` field points to a priority list of nodes of the new type `futex_h`. Each such node corresponds to a task (`futex_h.task`) added as helper to the futex whose key is equal to `futex_h.key`. Our introduced mechanism PI-CV needs to propagate the priority of the top-priority waiter task of a condition variable, to all the helpers with current lower

DOI: 10.1002/spe

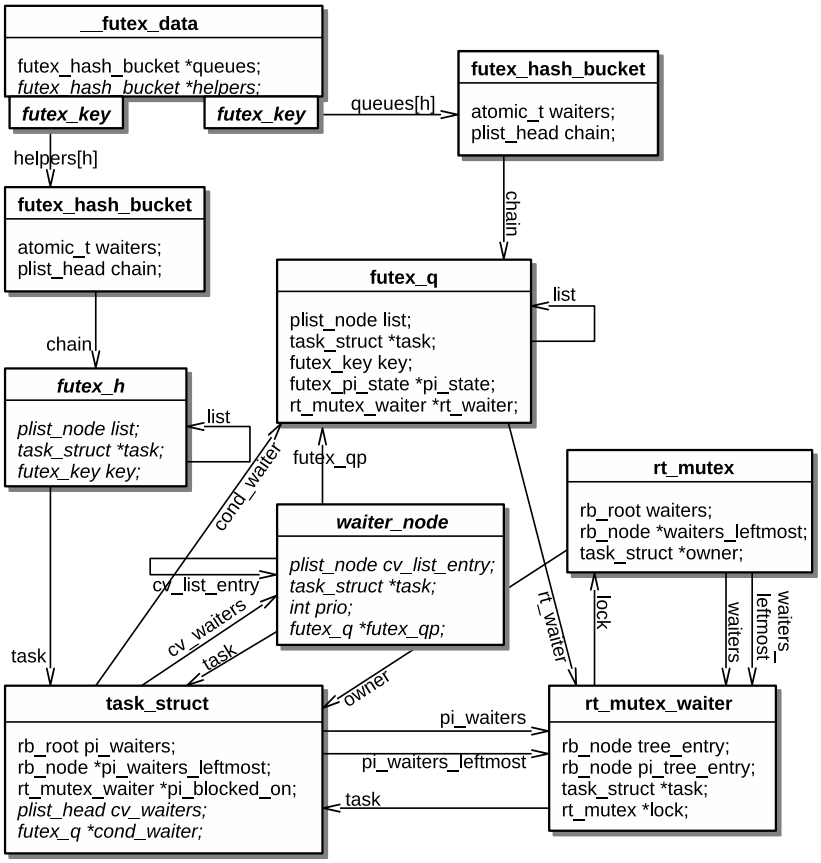

Figure 4. Summary of major changes to in-kernel data structures.

priority. This is done by simply scanning through the `__futex_data.helpers[h]->chain` list.

Whenever a task suspends on a condition variable, its priority may potentially have to be propagated through the whole chain of suspensions due to both futexes and rt-mutexes, till we encounter a ready task, which inherits the suspending task priority if higher than its own. In order to enable this walkthrough, we added within the task descriptor `task_struct` the `cond_waiter` field to point to the `futex_q` node identifying the condition variable this task is suspended on, and at the same time pointing back to the task.

In order to explain how priority inheritance has been realized supporting the transitive propagation of priority across futexes and rt-mutexes, we need to recall shortly how priority inheritance is handled for regular rt-mutexes in the kernel.

The `task_struct` task descriptor had already a red-black tree (RBT) of top-priority waiters blocked on rt-mutexes held by the task, pointed to by the `pi_waiters` and `pi_waiters_leftmost` fields; this enables fast retrieval of the highest priority task among tasks blocked waiting for any of the mutexes held by a task, so as to realize the priority inheritance protocol on rt-mutexes. The `rt_mutex` structure itself has also `pi_waiters` and `pi_waiters_leftmost` fields, constituting a RBT of all the tasks blocked on the mutex. Only the top-priority task among those waiting for a rt-mutex, is also inserted into the RBT of the waiters associated to the task owning the mutex lock (`rt_mutex.owner`). Finally, the `task_struct.pi_blocked_on` field points to the rt-mutex the task is currently blocked on, if any, through a `rt_mutex_waiter` node. Each such node associates a rt-mutex with one of the tasks waiting for it to be released. The same `rt_mutex_waiter` node can be inserted into both a task and a rt-mutex RBT structures (through the `tree_entry` and `pi_tree_entry` fields).

DOI: 10.1002/spe

Similarly to the `pi_waiters` field, we added a new `cv_waiters` field to the task descriptor, pointing to a priority list of the top-priority tasks waiting on condition variables this task is helping. This is done by linking in the priority list nodes of the new added type `waiter_node` (through the `task_struct.cv_list_entry` field). Each such node points back to the `futex_q` node (`waiter_node.futex_qp`) of one of the helped condition variables, corresponding to the top-priority waiter task.

For further details about the design of internal data structures of the Linux kernel, please, refer to the official documentation and source-code.

4.3. How it works

Helpers can be associated to a CV using the `sys_futex()` syscall. We added a new operation to this syscall, called `FUTEX_COND_HELPER_MANAGE`, with which a helper for a CV can be added or removed. Actual arguments are the CV user-space address (`u32 *`), helper's Linux thread ID (TID, as a `u32`) and a flag to switch between add or remove operations (`u32`).

Once a CV got helpers associated, any waiter that blocks on it can trigger the inheritance mechanism. The core of the implementation resides in two functions, `task_blocks_on_condvar()` and `task_wakes_on_condvar()`. The former is called before a waiter task `w` is actually put to sleep and is responsible to check if any of the helpers can inherit the waiter's priority. This is done through the following steps:

1. find the CV helpers hash bucket
2. for each helper task `h` in the bucket with `key` matching with the CV
   (a) get the old top-priority waiter task `old` for `h`, i.e., the first item in `h->cv_waiters`
   (b) insert `w` into `h->cv_waiters`, allocating a new `waiter_node` pointing to the `futex_q` node corresponding to the suspension of `w` on the CV
   (c) if the priority of `w` is strictly lower than the minimum among the helper's own priority and the priority of the old top-priority waiter `old`, then boost the current priority of `h` to the one of `w` and propagate calling `helper_adjust_prio(h, w)` (note that higher priorities in the kernel correspond to lower urgency/importance in the scheduler).

The function `task_wakes_on_condvar()` is called after a `pthread_cond_signal()` or `pthread_cond_broadcast()` for each waiter `w` that is going to be woken-up, and it performs dual operations w.r.t. the previous one:

1. find the CV helpers hash bucket
2. for each helper task `h` in the bucket with `key` matching with the CV
   (a) remove `w` from `h->cv_waiters`, disposing of the associated `waiter_node`
   (b) if `h->prio` was boosted due to `w` waiting on the CV, then deboost the current priority of `h` to the minimum among its own priority and the one of the new top-priority waiter, and propagate calling `helper_adjust_prio(h, w)`.

The mentioned `helper_adjust_prio(h, w)` function is used to propagate boosting or deboosting of the priority of `w` to all the tasks reachable following all the blocking chains from `h` forward; while doing so, a recursive call to `helper_adjust_prio()` is done to propagate inheritance due to a block on another condition variable, whilst a call to `rt_mutex_adjust_prio_chain()` is done to propagate inheritance due to a block on a rt-mutex. The rt-mutex existing code has also been slightly changed. For example, the `rt_mutex_getprio(t)` function was previously returning the minimum (lower priority values correspond to higher urgency in the scheduler) among the task own priority `t->normal_prio` and the priority of the top-waiter task `t->pi_waiters_leftmost`; now this function computes a 3-way minimum considering also the priority of the top-priority waiter task due to blocking on condition variables, found as the first node in the `t->cv_waiters` priority list.

DOI: 10.1002/spe

4.4. Theoretical overheads

Overheads due to PI-CV are clearly related to the number of declared helpers and the possible blocking patterns that can occur at run-time. We can distinguish a set of major contributions to PI-CV overheads, depending on the operation being performed, i.e., block due to a wait on a CV, wake-up due to a signal, addition or removal of a helper. In the following, for the sake of clarity and simplicity, we assume that hash-based look-ups of tasks and helpers within condition variables hash buckets, as well as within the tasks table, and memory allocation and deallocation operations, have all a constant complexity $O(1)$.

Whenever a task w blocks on a condition variable, in order to check whether to apply priority inheritance, the list of helpers has to be walked, and for each helper task with lower priority than the one of w, we need to: 1) inherit the priority, which is done by dequeueing the helper from its runqueue, and enqueueing it back with the new priority (note that each of these operations has a computational complexity that is constant for the real-time scheduling class and logarithmic for the CFS scheduler); 2) chain w into the helper's `task_struct.cv_waiters` priority list; 3) if the helper is blocked on its own (on another condition variable or on a rt-mutex), then propagate the inheritance. Let n denote the worst-case number of reachable tasks walking through the blocking chains including w at any given time. Then, the worst-case computational complexity is $O(n)$ if all involved tasks are under the real-time scheduling policy.

For each waiter task w that is woken up due to a signal on a CV, we need to scan again the list of helper tasks, and, for each helper that was inheriting the priority of w, we need to: 1) restore its original priority cancelling the effect of the inheritance; 2) unlink w from the helper's `task_struct.cv_waiters` priority list; 3) propagate the inheritance cancellation. This is again an $O(n)$ complexity.

When adding a helper to a condition variable, if no task is waiting for the condition, then we just need to insert the helper into `__futex_data.helpers[h]->chain` hash bucket priority list, where h is the hash of the condition variable key. This results in a linear complexity with the number of helpers associated to the condition variable. On the other hand, if at least one task is already waiting on the condition variable when a helper is added, then additional actions are needed for inheritance propagation, if the new helper has lower priority than the priority of the highest-priority blocked task, resulting in a $O(n)$ complexity, similar to the task blocking case.

When removing a helper from a condition variable, if no task is waiting for the condition, then we have a simple node removal from the priority list of helpers, with a linear complexity in the number of helpers. If at least one task is waiting instead, then if the helper being removed was boosted due to PI-CV, then its priority has to go back to its original value, and we may need to propagate priority adjustments, resulting in a $O(n)$ complexity as above.

Note that, in a real scenario: a) the number of helpers is often expected to be 1 or a very small number, because the use of PI-CV is recommendable in settings where it is clear throughout the application logic what is the task that is in charge of helping a given condition variable; b) the blocking chains should be very short, ideally at most 2-3 elements, and so the worst-case blocking graph to be expected at run-time will not contain many elements. Therefore, our implementation is not optimized to reduce computational complexity in the number of helpers.

Finally, note that the standard Linux kernel support for priority inheritance within rt-mutexes among tasks belonging to different scheduling classes is incomplete [10, 11], notably since the introduction of the new `SCHED_DEADLINE` scheduler. Our current implementation of PI-CV is limited to handling tasks in the real-time class. A fully engineered implementation dealing with all the possible cases with tasks under different priority classes is out of the scope of the present work.

5. SCHEDULABILITY ANALYSIS

Similarly to the priority inheritance mechanism traditionally used by real-time operating systems to access shared resources, PI-CV helps in reducing unneeded priority inversion in certain scenarios. There are two main advantages. The first one is an improved responsiveness of interactive and

DOI: 10.1002/spe

soft-real-time applications: this can be quantified by measuring statistics on the response times distributions of the tasks, and we will discuss it in Section 6.2.

The second advantage is the possibility to perform a worst-case analysis to bound the response time of critical real-time tasks that cooperate through condition variables. The analysis depends on the task model, and on the type of cooperation synchronization pattern that the tasks use. In particular, unlike the priority inheritance protocol for mutex semaphores, in this case the worst-case blocking analysis depends on how these variables are used in the program, and a completely generic analysis is impossible. Therefore, in this section, we give an example of the schedulability analysis for a Remote Procedure Call (RPC) programming pattern.

We assume a real-time system consisting of n periodic *client* tasks $\mathcal{T} \triangleq \{\tau_1, \ldots, \tau_n\}$, and m *server* tasks $\mathcal{S} \triangleq \{S_1, \ldots, S_m\}$. A client task τ_i is a periodic task with priority p_i that every period T_i releases an instance (also called *job*) which performs some computation with worst-case execution time C_i[‡], to be completed within its next activation (i.e., the relative deadline of τ_i is equal to its period T_i). During its execution, the task invokes one or more *remote procedures*. Each remote procedure is implemented by a server task S_j, having a configured priority lower than the one of any client task, which is boosted via PI-CV every time a client makes a RPC. Each task τ_i performs K_i remote procedure invocations, where the k-th invocation is done to the $r_{i,k}$-th server, $S_{r_{i,k}}$, and requires a processing time with WCET of $D_{i,k}$. Each activation of a task τ_i requires an overall worst-case execution time E_i on the CPU that includes both local processing within τ_i and K_i remote procedures, resulting in: $E_i \triangleq C_i + \sum_{k \in \{1,\ldots,K_i\}} D_{i,k}$.

The analysis in this section assumes a single-processor system, but the same results apply also to partitioned multi-processor systems, where each task is pinned down on a specific processor, and each client task can make calls only to servers on the same processor.

Requests are served sequentially: we assume that each server has an incoming queue where client tasks enqueue their requests. We assume that the incoming queue is large enough to contain all requests from all clients, so that every time a client posts a request, there is at least one free position in the queue. We also assume that, for every client i and server j, there exists a data structure where the client waits for completion of the remote procedure invocation, and retrieves the result, if any. This is done using a condition variable $\mathsf{CV}_{i,j}$. The client sends the request to the server incoming queue, then it performs a wait operation on $\mathsf{CV}_{i,j}$.

Each server S_j is blocked waiting for requests to be pushed within its incoming queue. When a request arrives from a client τ_i, the server pulls it out of the queue and performs the requested procedure, which has a worst-case execution time $D_{i,j}$; when it completes, it sends the result to the corresponding data structure, it performs a signal on $\mathsf{CV}_{i,j}$, and returns checking its input queue. For simplicity, we assume that servers do not invoke other RPC operations on other servers. We assume that requests in the incoming queue of each server are ordered by the priority of the corresponding client task. We apply our PI-CV protocol on each condition variable $\mathsf{CV}_{i,j}$: server S_j is set as the helper task for CV_j, so it inherits the priority of τ_i when the latter performs a wait on $\mathsf{CV}_{i,j}$. In Section 6 we will show how the server incoming queue and the client-server response data structure have been realized in the presented experiments.

Given the assumptions above, we apply the well-known response-time analysis (RTA) [12] to the set of client tasks, consisting in computing the worst-case response time R_i of each client task τ_i, from the highest priority to the lowest priority task, verifying that $R_i \leq T_i \, \forall \, \tau_i \in \mathcal{T}$.

With reference to a client task τ_i, it is convenient to introduce the set of higher-priority client tasks $\mathcal{T}_i^{hp} \triangleq \{\tau_j \in \mathcal{T} \mid p_j \geq p_i \wedge j \neq i\}$, and the set of lower-priority ones $\mathcal{T}_i^{lp} \triangleq \{\tau_j \in \mathcal{T} \mid p_j \leq p_i \wedge j \neq i\}$. [§]

The worst-case scenario for a client task τ_i is the one of synchronous activation with all higher priority client tasks $\mathcal{T}_i^{hp}$, in an instant in which lower-priority client tasks just submitted a request

[‡] The WCET refers to the time needed to complete each activation/instance excluding any time slice in which the CPU is preempted by other higher priority tasks in the system.

[§] The discussion is more easily followed thinking of a system with client tasks with pairwise distinct priorities. However, in those cases with more client tasks with the same priorities, these definitions allow to count same-priority tasks with their worst-case possible interference.

Prepared using speauth.cls
DOI: 10.1002/spe

to the same servers needed by τ_i. The response time can then be computed as the sum of the overall WCET E_i due to an activation of τ_i (including the WCET of its own RPCs, which execute at priority $\geq p_i$ thanks to PI-CV), plus the overall WCET E_j of all activations of higher priority tasks that may preempt τ_i (including the WCET of their RPCs, which execute at priority $\geq p_j \geq p_i$), plus the WCET of RPCs to servers inheriting a higher priority than τ_i, when these calls were issued just before τ_i's activation. This can be computed by finding the fixed point of the following iterative formula:

$$R_i = E_i + I_i + \sum_{j \in \mathcal{T}_i^{hp}} \left\lceil \frac{R_i}{T_j} \right\rceil E_j, \tag{1}$$

where $\lceil \cdot \rceil$ denotes the ceil operator and I_i is a term due to two contributions:

- the queueing time for RPCs made by τ_i to servers which are already serving a request from a lower priority task¶
- the time during which τ_i is preempted by higher-priority servers serving requests from lower-priority tasks.

The worst-case I_i can be computed by using two properties similar to the ones valid for priority inheritance on mutexes [6]:

- each lower priority task τ_j can contribute to I_i with at most one RPC;
- each server S_k can preempt a job of τ_i serving requests from lower priority tasks at most once.

First of all, we are going to prove these 2 properties.

Lemma 1
Any lower-priority task $\tau_j \in \mathcal{T}_i^{lp}$ can delay the execution of a job of task τ_i for at most the duration of a single server call from τ_j.

Proof
Since the priority of task τ_j is lower than the priority of task τ_i, τ_j has only 2 ways to delay the execution of τ_i: making a RPC to a server S_k with inherited priority $\geq p_i$, or using a server S_x that is also used by τ_i (so that the RPC from τ_i will incur queueing delay). The first situation happens when τ_j starts an RPC to S_k (not used by τ_i) before another task τ_h with priority $\geq p_i$ starts an RPC to the same server S_k. Hence, S_k inherits τ_h's priority and can preempt τ_i. The second situation happens when τ_j starts an RPC to S_x before τ_i starts its own RPC; hence, τ_i has to wait for τ_j's request to be served. In both cases, τ_j must start its RPC *before the job of* τ_i *arrives*; after this request has been served, τ_j is not able to execute until τ_i's job finishes (because if τ_i blocks again waiting for a server, the server will inherit τ_i's priority, which is larger than τ_j's one).
Hence, τ_j may delay the execution of τ_i for at most the duration of one RPC. □

Lemma 2
While a job of task τ_i is active, any server S_k can serve at most one request from any task τ_j with priority $p_j \leq p_i$.

Proof
This comes from the fact that the incoming requests queue of a server is ordered according to the client tasks' priorities: S_k can execute requests from lower priority tasks only if it is inheriting the priority of a higher priority task or if τ_i is blocked on an RPC on S_k while S_k is serving a lower priority task. In both cases, after the lower priority task's request is served S_k will not serve other requests from lower priority tasks (in the first case, it will serve requests from higher priority tasks, and in the second case it will serve τ_i's request). □

¶Notice that the preemption from servers serving RPCs from higher priority tasks is already accounted for in the E_j terms in Equation (1);

DOI: 10.1002/spe

Based on these two properties, it is possible to compute I_i as follows, after introducing some further notation. Let $\mathcal{K}_i$ denote the set of indexes of servers called by τ_i: $\mathcal{K}_i \triangleq \{k \in \{1,\ldots,m\} \mid \exists h \in \{1,\ldots,K_i\} \text{ s.t. } r_{i,h} = k\}$. Let $\mathcal{Q}_i$ denote the set of server calls (j,k) made by any lower priority task τ_j to any server S_k that might cause queueing delay to τ_i:

$$\mathcal{Q}_i \triangleq \left\{(j,k) \mid \tau_j \in \mathcal{T}_i^{lp} \wedge k \in \mathcal{K}_i \cap \mathcal{K}_j\right\}. \tag{2}$$

Let $\mathcal{P}_i$ denote the set of server calls (j,k) made by any lower priority task τ_j to any server S_k that can also be called by any higher priority task τ_h:

$$\mathcal{P}_i \triangleq \left\{(j,k) \mid \tau_j \in \mathcal{T}_i^{lp} \wedge \exists \tau_h \in \mathcal{T}_i^{hp} \text{ s.t. } k \in \mathcal{K}_j \cap \mathcal{K}_h\right\}. \tag{3}$$

Then, I_i is obtained as the worst-case sum of WCETs of a subset of the calls referenced in $\mathcal{Q}_i \cup \mathcal{P}_i$, where, in each sum, a lower-priority task cannot appear more than once (due to Lemma 1), and a called server task cannot appear more than once (due to Lemma 2), i.e.:

$$\mathcal{W}_i^* \triangleq \left\{\mathcal{A} \subseteq \mathcal{Q}_i \cup \mathcal{P}_i \text{ s.t. } \forall (j,k) \in \mathcal{A}, \begin{array}{l} \left|\{(\tilde{j},\tilde{k}) \in \mathcal{A} \text{ s.t. } \tilde{j} = j\}\right| = 1 \\ \wedge \left|\{(\tilde{j},\tilde{k}) \in \mathcal{A} \text{ s.t. } \tilde{k} = k\}\right| = 1 \end{array}\right\} \tag{4}$$

$$I_i = \max_{\mathcal{A} \in \mathcal{W}_i^*} \sum_{(j,k) \in \mathcal{A}} \max\{D_{j,h} \mid r_{j,h} = k\}, \tag{5}$$

where $|\cdot|$ in Equation (4) denotes the set cardinality operator, and the rightmost $\max$ in Equation (5) is due to the fact that, in theory, a task j could make multiple calls to the same server k, so we need to consider the worst-case call. Note that the formula for I_i obtained above is similar to the formula for computing the blocking time of the Priority Inheritance Protocol for non-nested critical sections [6, 13].

6. EXPERIMENTAL EVALUATION

In this section, we provide extensive experimental validation of our proposed PI-CV mechanism, using the implementation in the Linux kernel as described in Section 4. The experiments have been run on an Intel®Core™ i7-4790K CPU at 4GHz (4 hyper-threaded cores, visible as 8 CPUs on Linux), with frequency locked at the maximum, running Linux Fedora 25 with a 4.10.0-rc3 kernel, modified with PI-CV.

First, we show a simple experiment validating the correct behavior of our implementation. Then, we highlight the benefits of using the mechanism on a synthetic application scenario, and finally we show what additional overheads PI-CV adds to the system when it is used under various stress conditions.

In the experiments that are described below, a common synchronization module has been re-used, realizing a synchronized finite priority queue between senders (or producers) and receivers (or consumers). This is a minimalistic variant of a standard textbook implementation of a synchronized finite queue, whose code is summarized in the listing in Figure 5, where for the sake of brevity we omitted error handling code, header/interface files, and the actual implementation details for the priority queue (`prqueue`). The synchronized queue has an initialization method `queue_init()` where one specifies the maximum queue size `qsize`. The queue is protected by the `q->mutex` rt-mutex (to ensure mutual exclusion during queue operations). The `queue_push()` operation blocks on a full queue, suspending the calling task on the `q->less` condition variable. Similarly, the `queue_pop()` operation blocks on an empty queue, suspending the calling task on the `q->more` condition variable. The queue can optionally take advantage of the PI-CV mechanism presented in this paper, when the `queue_add_producer()` and `queue_add_consumer()` functions are called after the `queue_init()`, so that the OS/kernel knows which tasks are expected to signal on each of the used condition variables.

Prepared using speauth.cls
DOI: 10.1002/spe

```
// Push an item into the queue (block if full)
void queue_push(queue_t *q, qitem_t item, int pr) {
  pthread_mutex_lock(&q->mutex);
  while (prqueue_full(&q->prqueue))
    pthread_cond_wait(&q->less, &q->mutex);
  prqueue_push(&q->prqueue, item, pr);
  pthread_cond_signal(&q->more);
  pthread_mutex_unlock(&q->mutex);
}

// Initialize queue with maximum given size
void queue_init(queue_t *q, int qsize) {
  prqueue_init(&q->prqueue, qsize);

  /* Initialize mutex and cond vars */
  pthread_mutexattr_t mutex_attr;
  pthread_mutexattr_init(&mutex_attr);
  pthread_mutexattr_setprotocol(
    &mutex_attr, PTHREAD_PRIO_INHERIT);
  pthread_mutex_init(&q->mutex, &mutex_attr);
  pthread_mutexattr_destroy(&mutex_attr);
  pthread_cond_init(&q->more, NULL);
  pthread_cond_init(&q->less, NULL);
}
```

```
// Pop an item out of the queue (block if empty)
qitem_t queue_pop(queue_t *q) {
  qitem_t item;
  pthread_mutex_lock(&q->mutex);
  while(prqueue_empty(&q->prqueue))
    pthread_cond_wait(&q->more, &q->mutex);
  item = prqueue_pop(&q->prqueue);
  pthread_cond_signal(&q->less);
  pthread_mutex_unlock(&q->mutex);
  return item;
}

// Add producer thread for queue
void queue_add_producer(queue_t *q, pid_t prod) {
  pthread_cond_helpers_add(&q->more, prod);
}

// Add consumer thread for queue
void queue_add_consumer(queue_t *q, pid_t cons) {
  pthread_cond_helpers_add(&q->less, cons);
}
```

Figure 5. Shared queue implementation taking advantage of the proposed PI-CV mechanism when `queue_init_helpers()` is called.

In order to highlight the advantages of our presented technique, producer and consumer tasks, as well as annoyer tasks, are all pinned down on the same CPU, and they are scheduled using the POSIX real-time scheduling class on Linux, using different real-time priorities, as detailed in each experiment.

In experiments taking advantage of the presented PI-CV mechanism: tasks that are known to push elements (producers) in the shared queue are added at the beginning as *helpers* for the `more` CV (they signal on it after having added a new element); tasks that are known to pop elements (consumers) are added as helpers for the `less` CV (they signal on it after having removed an element).

6.1. Runtime validation

We used a synthetic benchmark implementing the classical *producer*(s) - *consumer*(s) scenario on a finite size queue of elements. An additional set of periodic, middle-priority *annoyer* threads is used to check if the inheritance mechanism works. The benchmark creates a specified number of producers and consumers that work on the same finite-size queue. Each of these two types of threads runs for a random amount of time (between 10ms and 100ms) each time they are activated. It is furthermore possible to specify a number of annoyers, with priorities higher than producers and lower than consumers, that activate and execute periodically (exact parameters are detailed below in each experiment).

We performed a simple test to validate the implementation. In the first test we ran the benchmark with one producer (`Prod`), one consumer (`Cons`) and one annoyer (`Annoy`). PI-CV can be enabled or disabled at start-up. In Figure 6 we show a visual representation [‖] of the threads execution with and without it.

When PI-CV is disabled (top sub-figure), `Annoy` can preempt `Prod` at any time instant in which it starts running, like the one denoted as `P` in the plot. Since `Cons` is blocked waiting for Prod, which is preempted by Annoy, we have priority inversion (`Cons` is actually waiting for the lower priority thread `Annoy` to finish execution) until instant `F`, when `Annoy` terminates. On the contrary, when PI-CV is active (bottom sub-figure), `Cons` donates its priority to `Prod` when it blocks calling `pthread_cond_wait()` on the PI-aware CV (upward red arrow). At

[‖] Execution diagrams in this section are created through the KernelShark (`https://lwn.net/Articles/425583/`) utility from execution traces extracted from the kernel via ftrace (`Documentation/trace/ftrace.txt`).

DOI: 10.1002/spe

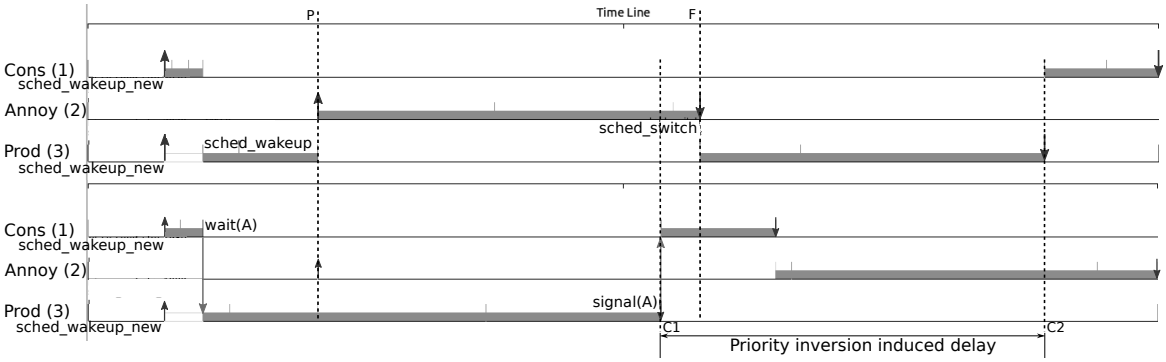

Figure 6. One producer (priority 3), one annoyer (priority 2) and one consumer (priority 1). PI-CV disabled (top figure) and enabled (bottom figure).

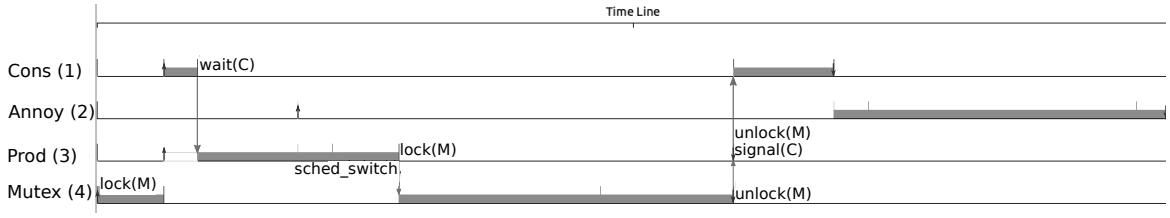

Figure 7. One producer (priority 3), one annoyer (priority 2) and one consumer (priority 1). Mutex thread (priority 4) shares rt_mutex M with producer. PI-CV enabled.

time `P`, `Prod` is not preempted by Annoy, since now has priority 1. When Prod terminates it calls `pthread_cond_signal()`, wakes `Cons` up and returns to its original priority (downward red arrow). `Cons` starts executing, since now the condition is true. `Annoy` can resume execution only after `Cons` is done. As a consequence, a priority inversion of duration `C2-C1` was avoided.

We performed a second validation test slightly modifying the benchmark. In this second test we wanted to prove that PI-CV can inter-operate with stock rt_mutex priority inheritance mechanism. We added another thread (called Mutex), to the application that shares some variable with producer. Mutual execution on the shared variable is achieved through the use of an `rt_mutex`. Figure 7 shows an execution in which PI-CV is enabled (we omit the non PI-CV case for space reasons).

The Mutex thread starts execution before the others and locks mutex `M`. It is then preempted by the consumer, that has higher priority. The consumer blocks on the PI aware condition variable and the producer starts to execute. The annoyer is ready to run in the middle of the producer execution, but it cannot perform preemption since the consumer donates its (higher) priority to the producer. When the producer tries to lock mutex `M` it has to wait, as the Mutex thread is holding the mutex. At this time the producer has (inherited) priority 1, and it gives this priority to thread Mutex that can resume execution (since consumer and producer are blocked and the annoyer has lower priority). If the PI-CV mechanism were not integrated with the standard `rt_mutex` priority inheritance, Annoyer could have resumed and delayed Mutex by an unbounded amount of time (causing a domino effect on producer and consumer). Original priorities are taken back after `unlock(M)` by thread Mutex and `signal(C)` by the producer.

6.2. Impact on Response Times

To better check the correctness of the the implementation presented in this paper, we implemented the client-server interaction scenario analyzed in Section 5 and we compared the theoretical analysis with the measurement performed on the real implementation. We tested multiple tasksets, verifying that the experimental results were always compatible with the theoretical expectations.

DOI: 10.1002/spe

Parameter	Client1	Client2	Annoyer	Server
Task period	40ms	50ms	60ms	-
RT priority	90	80	70	50
Comp. time	10ms	10ms	10ms	4.5ms

Table I. Task parameters for the scenario.

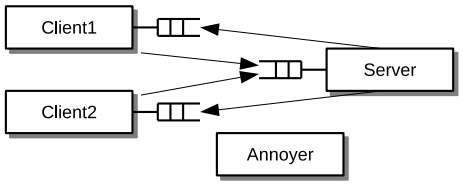

Figure 8. Scenario under consideration.

To highlight the advantages that PI-CV brings, here we report the results obtained in a simple testcase (see Figure 8) with 2 client tasks running at different real-time priorities, making calls to the same server task. In the scenario, each client task is periodic: it spends some time processing, it invokes the server by pushing a message onto the server receive queue, then it waits for a response to be placed by the server onto the client's own receive queue. The server waits for an incoming message on its receive queue, then it computes for some time, then it pushes a message back onto the receive queue of the caller task, and it repeats forever. All queues are enhanced by PI-CV as described above and helper tasks are declared so that the empty-queue condvar `more` of each client receive-queue has the Server as helper, enabling donation of their priority to the Server whenever waiting for it to deliver them a response to a call.

Table I summarizes the parameters used for all the tasks in the scenario. For example, Client1 wakes up every $40ms$, it computes for $10ms$, then it makes a call to Server, where its request needs $4.5ms$ of computing time before an answer is delivered to Client1, that goes to sleep until the next instance. The server uses the enqueued integer on its incoming receive queue to distinguish among calling clients, and to identify the queue onto which the response has to be pushed.

In a first scenario, the server has been assigned a static RT priority of 50, the lowest among all the real-time tasks within the scenario. A further task, called Annoyer, is also periodic, but it does not interact with others. Its priority is lower than clients, but higher than the server priority.

Note that the execution times mentioned in Table I are actually WCETs: the computation parts of the tasks are implemented as busy loops, executing a pre-fixed number of iterations tuned so as to match experimentally the reported/wanted WCETs. The resulting average computation times throughout the experiments have been slightly lower.

Again, notice that multiple different tasksets have been tested and the particular scenario reported here has been selected to easily highlight the advantages of PI-CV – the task periods reflect typical periods of multimedia applications, in the range of tens of milliseconds, and priorities of periodic tasks have been set according to a typical rate-monotonic assignment. The overall experiment duration has been set to 60 seconds, amounting to roughly 1200-1500 activations for the client tasks.

All tasks have been configured with a SCHED_FIFO scheduling discipline, and the RT priority as indicated above. Their affinity has been set to pin them down to the same CPU, to keep the experiment simple and its outcomes easily understandable. Similar results can be obtained with more tasks without setting their affinity. On a related note, the default 950ms cap for RT tasks on Linux (via RT throttling) has been disabled by writing -1 into $/proc/sys/kernel/sched\_rt\_runtime\_us$.

Two sets of experiments have been performed, one with only the traditional priority inheritance on all mutexes, and the other one with also PI-CV on all condition variables (the two mechanisms acted in an integrated fashion as explained above).

DOI: 10.1002/spe

	No PI-CV		PI-CV	
Statistic	Client1	Client2	Client1	Client2
Average	25.776	33.587	15.379	22.816
90th perc.	33.71	38.489	18.913	28.938
Maximum	33.76	38.55	18.995	28.996
Analytical Worst-Case	—	—	19	29

Table II. Response-time statistics (in milliseconds) for Client1 and Client2 in the scenario.

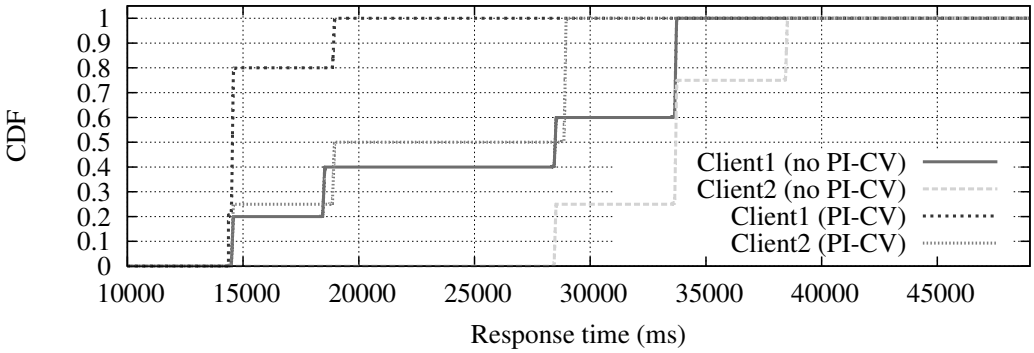

Figure 9. Response time CDFs of Client1 and Client2 in the two cases of with and without PI-CV.

Figure 9 reports the obtained cumulative distribution functions (CDFs) ** of the response time (i.e., the difference between the job finishing and arrival times) of the two clients, with and without PI-CV. The impact of PI-CV on the tasks performance is also quantified in terms of change in the average, 90th percentile and maximum observed response times for the two tasks, as visible in Table II.

It is clearly visible that, when using PI-CV, the response times of both the two clients (Client1 and Client2) are greatly reduced. This is due to the PI-CV mechanism allowing to avoid unnecessary priority inversion. For example, the highest-priority task (Client1), benefits from PI-CV with: a reduction of its worst-case response time by about 44% (from $33.76ms$ down to $18.995ms$) and a reduction in its average response time by about 40% (from $25.776ms$ down to $15.379ms$). The other client (Client2) also greatly reduces its worst-case and average response time.

But there is more than a simple reduction of the response times: using PI-CV, the real-time performance of the two clients become predictable, allowing to provide real-time guarantees as shown in Section 5. In fact, the maximum response times measured for the two clients (indicated as "Maximum" in the table) are consistent with the worst-case response times computed according to Equation (1) (indicated as "Analytical Worst-Case" in the table). For Client 1, the equation gives $R_1 = 10 + 4.5 + 4.5 = 19ms$, since $C_1 = 10ms$, $D_1 = 4.5ms$ and $I_1 = 4.5ms$ (because a request from C_1 can arrive when the server just started to serve a request from C_2) and there is no interference from higher priority tasks. The measured maximum response time ($18.995ms$) is consistent with this result. For Client 2, the equation gives $R_2 = 10 + 4.5 + 14.5 = 29ms$, since $C_2 = 10ms$, $D_2 = 4.5ms$, $I_2 = 0ms$ (because there are no tasks with priority lower than Client 2) and the interference from Client 1 is equal to $C_1 + D_{1,1} = 10 + 4.5 = 14.5ms$. In this case, the worst-case situation is reproduced in our test-case, and the measured maximum response time is about the same as the worst-case computed according to the theoretical analysis.

In the above experiments, the server task has been run as the lowest priority task in the system. Therefore, when a high-priority task in the system calls the server, it is subject to interference by

**Note that the upper bound for the plots has been stretched to 1.05, just to visually highlight the maximum observed response-time for the various curves, that was non-visible otherwise.

Prepared using speauth.cls
DOI: 10.1002/spe

lower-priority tasks (other clients or the annoyer) whose priority is higher than the server. PI-CV remedies to the problem by raising dynamically the priority of the server as needed, when needed. An alternative approach would have been to assign a higher priority to the server task, so that it can immediately execute as soon as a client submits a request. This approach, which is similar to *Immediate Priority Ceiling* [6], requires to know the priorities of all the clients in advance. Such a knowledge is needed because the priority of the server has to be as large as the priority of the highest priority real-time client; however, if the server's priority is too high the server risks to preempt unrelated real-time tasks, increasing their response times (and generating other priority inversions). On the other hand, with PI-CV the priority of server tasks is automatically and optimally tuned by the operating system (as opposed to having to be manually tuned), simplifying the design of real-time applications composed by multiple interacting tasks.

It has to be noted that the effectiveness of PI-CV and its quantitative impact on the tasks performance depends essentially on how much time a task spends wait()-ing on a condition variable for which helper tasks are defined, i.e., how much time is needed for the corresponding signal() to occur. This time is of course very application-specific. Comparing with traditional priority inheritance on mutex semaphores, in that case the effectiveness of the mechanism depends on how much time a task spends in a critical section with a mutex locked, which is *also* very application-specific. Though, the time spent with a mutex locked may be expected to be lower than the one spent wait()-ing for a signal() by some other task. Therefore, whenever it is possible to identify dependency relationships among real-time tasks, the presented PI-CV mechanism may be exploited to avoid situations of priority inversion expected to generally be of longer duration, compared to situations in which only priority inheritance on mutexes is used.

6.3. PI-CV overheads

In this section, we perform an experimental evaluation of the runtime overheads comparing the Linux kernel patched with PI-CV modifications to a vanilla one. All the experiments have been performed using the same hardware and software configurations described in Section 6.1.

We measured the runtime overheads comparing execution of the same benchmark as introduced in Section 6.1 with and without PI-CV. We measured kernel functions duration, using the ftrace infrastructure, when the mechanism is enabled and when not. We run the benchmark several times, for 60 seconds each time, varying the number of running threads. Producer and consumer threads behave similarly. Producers lock the mutex associated to the CV, check if space is available in the queue and wait for it if not, run for certain amount of time comprised between 10ms and 100ms (they logically produce new data in this interval), signal that a new item is ready, unlock the mutex and sleep for one second. Consumers lock the mutex, wait for new data (they block on the PI-aware CV boosting producers), when woken up they consume an item (running for an interval between 10ms and 100ms), signal that a slot is empty and release the mutex. Annoyers execute for 300ms and then sleep for a second.

Every kernel function can be profiled through `ftrace`. We report here the duration measurements of only six of them, those that could be ill-affected by PI-CV. We provide measurement plots for the functions:

a) `do_futex()`, that is the function performing `sys_futex()` demuxing;
b) `futex_lock_pi_atomic()`, that implements the atomic work required to acquire a PI aware futex;
c) `futex_requeue()`, that is the function invoked by a `pthread_cond_signal()` and includes time spent on `task_wakes_on_condvar()`;
d) `futex_wait_queue_me()`, that is the function invoked by ta `pthread_cond_wait()` and includes time spent on `task_blocks_on_condvar()`;
e) `task_blocks_on_condvar()`, that is a new function potentially boosting helpers and starting priority inheritance propagation;
f) `task_wakes_on_condvar()`, that is new function responsible for revoking priority boosting if necessary.

DOI: 10.1002/spe

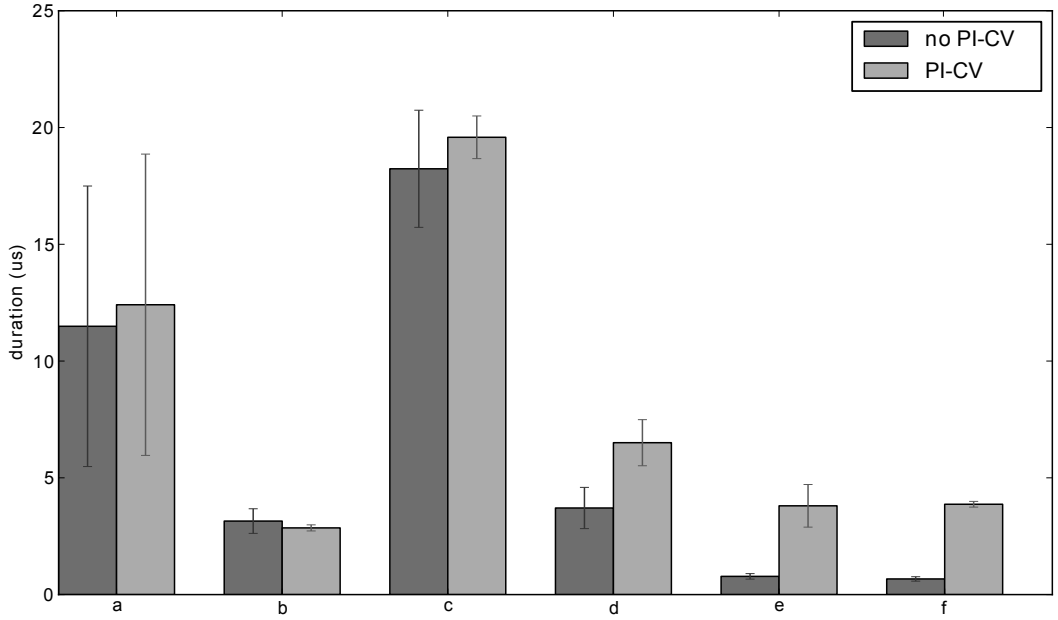

Figure 10. Time consumed by `do_futex()` (a), `futex_lock_pi_atomic()` (b), `futex_requeue()` (c), `futex_wait_queue_me()` (d), `task_blocks_on_condvar()` (e) and `task_wakes_on_condvar()` (f) with a producer, an annoyer and a consumer.

Figure 10 shows the simplest case (one producer, one consumer and one annoyer). In this case the overheads are small, since the duration of the functions in the PI-CV case is slightly higher than without the mechanism. The functions that implement priority boosting last considerably longer when PI-CV is enabled, but they do not affect the syscall duration as they represent a small fraction of it. Figure 11 shows results with one producer, three consumers and two annoyers. The number of consumers waiting on a PI-aware CV do not increase syscalls duration. It is instead interesting to note (Figure 12) that overheads increase with an increase in the number of producers, as expected due to the reasons explained in Section 4.4.

We performed a further experiment to highlight the growth of overheads with the number of helpers: we kept a single high-priority producer with a $10ms$ activation period, and a variable number of low-priority consumers between 1 and 16. All consumers are registered as helpers for the client blocking call. The consumers immediately reply back to the blocked client, in order to gather the pure operating system overheads, due essentially to execution of system calls, task scheduling, context switch. Among the PI-CV theoretical overheads introduced in Section 4.4, we are focusing here on the sum of the overheads due to a task blocking on a wait, and being woken up with a signal. In Figure 13, we report the average and standard deviation of the ping-pong times as observed by the client task, over a 2 minutes run, where 24000 calls are done. Overheads grow with the number of helpers roughly at a rate of $0.4\mu s$ per additional helper. Note that this scenario is designed so that all the low-priority helper tasks actually inherit the priority of the higher-priority producer making the blocking request. What contributes to the measured overheads in this case, is the actual action of changing the real-time priority of a task in the kernel, which is done for all the $1, 2, \ldots, 16$ helper tasks in the scenario. However, in a real settings, whenever a task blocks on a condition variable, the blocked tasks DAG is only visited as long as lower-priority tasks are met, in which case their priority needs to be changed. Whenever encountering tasks with already the same or higher priority than the blocking task, the corresponding blocking chains walk can terminate earlier, saving time.

We finally performed a series of runs relaxing the single-core assumption, using all the 8 CPUs available on the platform. In Figure 14 we show that the overheads are still comparable in one of this

Prepared using speauth.cls
DOI: 10.1002/spe

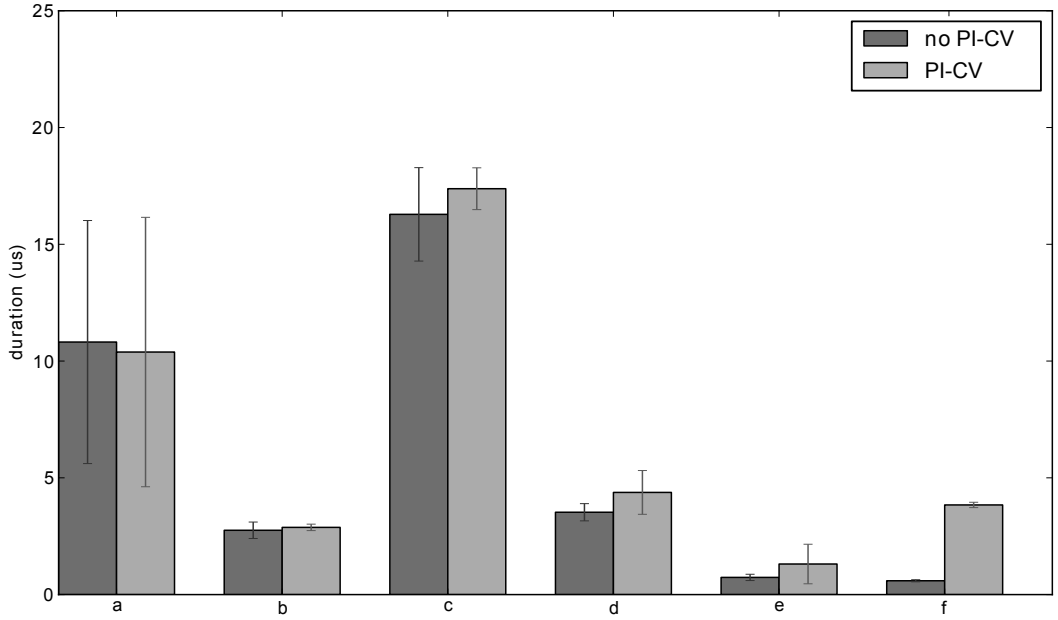

Figure 11. Time consumed by `do_futex()` (a), `futex_lock_pi_atomic()` (b), `futex_requeue()` (c), `futex_wait_queue_me()` (d), `task_blocks_on_condvar()` (e) and `task_wakes_on_condvar()` (f) with a producer, two annoyers and three consumers.

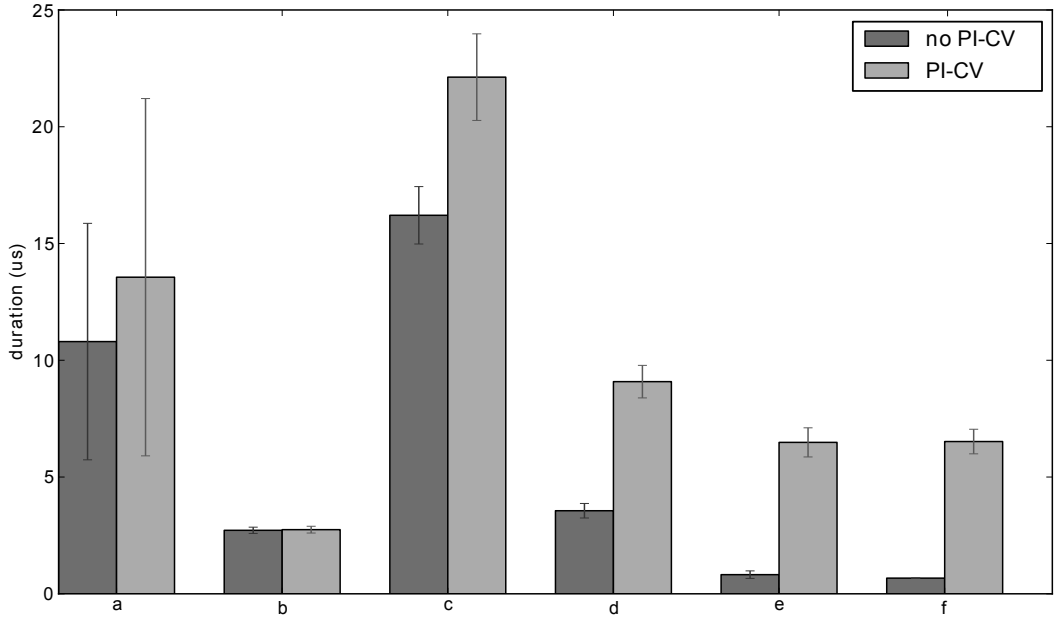

Figure 12. Time consumed by `do_futex()` (a), `futex_lock_pi_atomic()` (b), `futex_requeue()` (c), `futex_wait_queue_me()` (d), `task_blocks_on_condvar()` (e) and `task_wakes_on_condvar()` (f) with three producers, two annoyers and one consumer.

extreme cases (10 producers, 4 consumers and 2 annoyers). Since the load is spread across available processors, helpers boosting chains are kept short resulting in low overheads.

DOI: 10.1002/spe

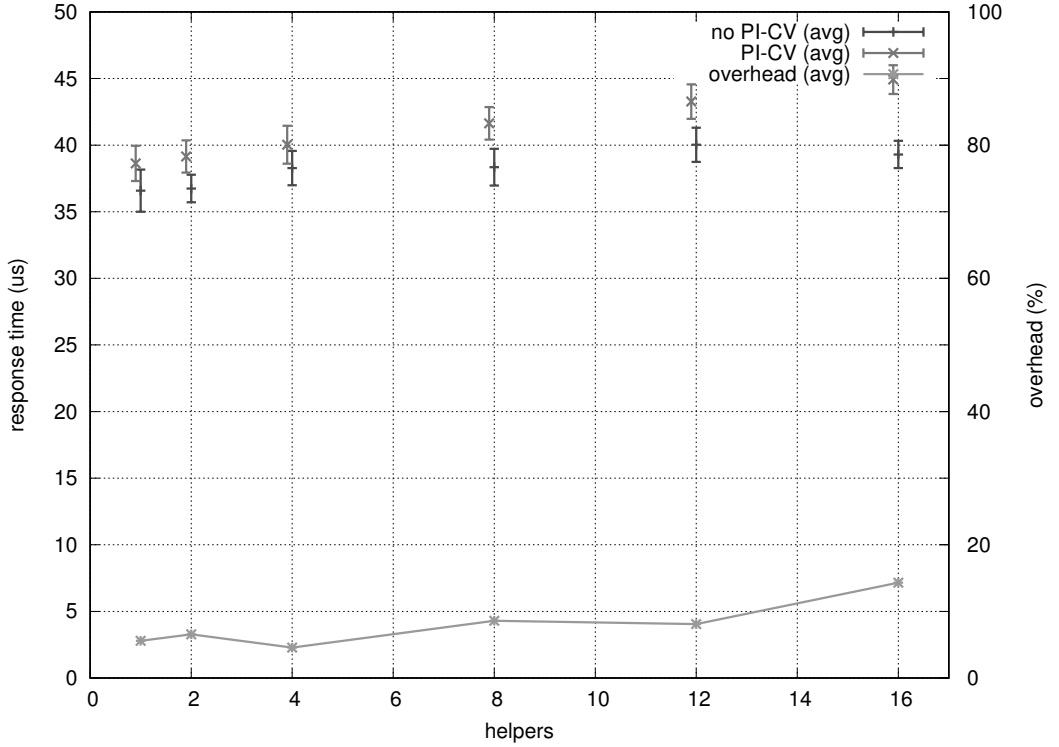

Figure 13. Ping-pong times (refer to the left Y axis) between a high-priority producer and a variable number (on the X axis) of low-priority consumers, with and without PI-CV (second and first curve, respectively). The relative increase in response times due to PI-CV is reported in percentage (third curve, refer to the right Y axis).

7. RELATED WORK

Although the priority inversion problem has been noticed earlier in 1980 [14], the first works investigating its impact in real-time systems date back to 1987, when Cornhill and Sha reported [15, 16] that, in the Ada language, a high-priority task could be delayed indefinitely by lower priority tasks under certain conditions, and formalized what are the correct interactions between client and server tasks in form of assertions on the program execution. Also, they introduced priority inheritance as a general mechanism for bounding priority inversion. Later, Sha et al. [6] formalized the two well-known Basic Priority Inheritance (BPI) and Priority Ceiling (PCP) protocols. While BPI allows a task to be blocked multiple times by lower priority tasks, with PCP a task can be blocked at most once by lower-priority tasks, so priority inversion is bounded by the execution time of the longest critical-section of lower-priority tasks; also, PCP prevents deadlock. Also, Locke and Goodenough discussed [17] some practical issues in applying PCP to concrete real-time systems.

Various extensions to PCP have been proposed, for example to deal with reader-writer locks [18], multi-processor systems [19, 20] and dynamically recomputed priority ceilings [21]. Furthermore, Baker introduced [22] Stack Resource Policy (SRP), extending PCP so as to handle multi-unit resources, dynamic priority schemes (e.g., EDF), and task groups sharing a single stack. More recently, Lakshmanan et al. [23] further extended PCP for multi-processors grouping tasks that access a common shared resource and co-locating them on the same processor. Schmidt et al. investigated [24] on various priority inversion issues in the CORBA middleware, and proposed an architecture (TAO) mitigating them.

When scheduling under the Constant Bandwidth Server (CBS) [25], Lamastra et al. proposed [26] the BandWidth Inheritance (BWI) protocol, allowing a task owning a lock on a mutex not only to inherit the (dynamic) priority of the highest priority waiting task (if higher than its own), but also

DOI: 10.1002/spe

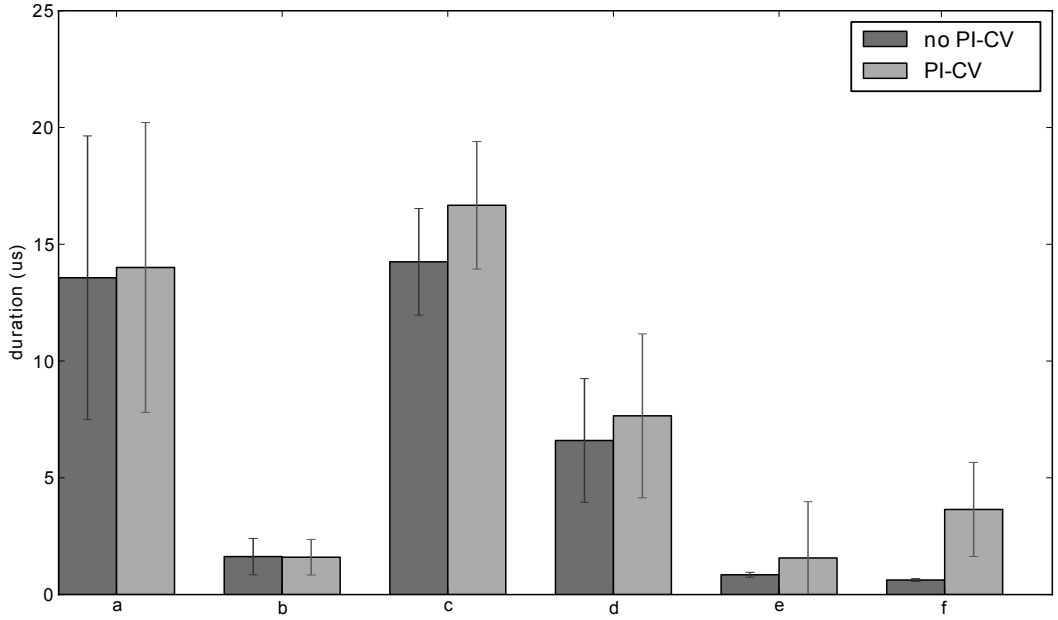

Figure 14. Time consumed by `do_futex()` (a), `futex_lock_pi_atomic()` (b), `futex_requeue()` (c), `futex_wait_queue_me()` (d), `task_blocks_on_condvar()` (e) and `task_wakes_on_condvar()` (f) with 10 producers, 4 consumers and 2 annoyers. Threads are free to execute on any out of 8 CPUs.

to account for its execution within the reservation of the task whose priority is being inherited. This allows to keep the temporal isolation property ensured by the CBS, in the sense that non-interacting task groups cannot interfere on each other's ability to meet their timing constraints. Later, Faggioli et al. [27] discussed issues and optimizations in the implementation of the protocol in the Linux kernel, and extended BWI to multi-processors [28].

Block et al. proposed FMLP [29], a resource locking protocol for multi-processor systems allowing for unrestricted critical-section nesting and efficient handling of the common case of short non-nested accesses. Guan et al. dealt [30] with real-time task sets where interactions among tasks are only known at run-time depending on which particular branches are actually executed.

Many other works exist in the literature [31, 32, 33, 34, 35, 36] on variants of the above resource-sharing protocols and their analysis. A comprehensive overview and comparative evaluation of them can be found in the recent work by Yang et al. [37].

Although the previously mentioned works focus on priority inversion due to mutual access to shared resources, some works also applied some form of inheritance in different contexts. For example, techniques to mitigate priority inversion have been applied in the context of scheduling virtual machines communicating with each other [38]. Other works considered client-server interactions between tasks, applying some form of inheritance [39]. For example, BWI can also be adapted to trigger inheritance when a client blocks waiting for the server's response [7], allowing to perform a schedulability analysis for that particular type of scenario. Also noteworthy is the proposed set of modifications to the Android Binder framework to preserve the nice level of the calling thread across remote procedure calls (RPCs) [2], extending the Binder standard capability to inherint nice levels across synchronous RPC calls[††].

The mechanism being presented in this paper is generic and can be used with custom inter-thread communications: while the other mechanisms focus on mutexes or client-server interactions, PI-CV

[††] For details, refer to the source code available at: `https://android.googlesource.com/kernel/common/+/android-4.9/drivers/android/binder.c`.

DOI: 10.1002/spe

is useful every time tasks interact via a generic blocking interaction model, such as made possible through the use of condition variables associated with mutexes. These are generally used in the implementation of custom shared data types supporting custom communication and synchronization protocols in concurrent systems. To the best of our knowledge, there are no alternatives for dealing with the specific type of problem of priority inversion as described above, in presence of CVs. Commonly known alternatives to using mutexes and locks at all, include recurring to lock-free data structures, and solutions based on the Transactional Memory programming paradigm [40]. Lock-free programming is well-known to be more complex and difficult to master, than traditional lock-based programming. On the other hand, the Transactional Memory programming paradigm is particularly useful in presence of non-blocking operations on shared data structures, thus it does not constitute an alternative to the presented technique. A thorough and detailed comparison among these communication and synchronization techniques is outside the scope of this paper.

Finally, note that Hart and Guniguntala [9] made changes to the GNU libc `pthreads` library and kernel in order to support efficient wake-up of multiple tasks waiting on a CV (as due to a `pthreads_cond_broadcast()`) used in connection with an rt-mutex, so as to avoid the "thundering herd" effect, and guaranteeing the correct wake-up order (considering also priority inheritance). Such changes relate to the support for priority inheritance in rt-mutexes and they are not to be confused with the mechanism being proposed in this paper.

8. CONCLUSIONS AND FUTURE WORK

In this paper, a mechanism has been presented for enhancing responsiveness of real-time software components in computing systems. It allows for explicitly declaring task dependencies that enable the OS to properly trigger priority inheritance on blocking interactions. An implementation of the mechanism has been presented for the Linux kernel and evaluated, demonstrating viability of the approach, leading for example to a 31% reduction in the worst-case response-time for the highest-priority task in a synthetic scenario. Also, a preliminary analysis technique has been shown that allows for ensuring schedulability of a set of interacting periodic real-time tasks.

In the future, we plan to extend this work along various directions. First, we would like to explore applicability of the presented concepts to interactions that do not necessarily rely on condition variables. Second, we would like to demonstrate the usefulness of the mechanism in real life, modifying existing applications. To this regard, it may be useful to integrate PI-CV within reusable frameworks that expose higher level abstractions for inter-task communications. For example, the well-known Apache Runtime Libraries (APR) expose already a synchronized queue abstraction leveraging condition variables, that may conveniently be extended to integrate PI-CV. In terms of application use-cases, an interesting scenario may be the one of multiple real-time applications communicating through a common publish-subscribe messaging daemon in a service-oriented architected embedded system [41]. Third, we would like to apply the technique under resource reservations scheduling along the lines of BWI [26, 42], e.g., integrating the current implementation with the `SCHED_DEADLINE` [43] CBS/EDF-based scheduler, that is integrated within the mainline Linux kernel since version 3.14. Also, it would be interesting to see how PI-CV could be integrated within the set of real-time enhancements to Android recently presented in [1].

The schedulability analysis method shown in this work was limited to a particular subset only of the possible interactions among tasks. In the future, we plan to extend the method to consider more general scenarios, for example when there are arbitrary DAG of dependencies among tasks, e.g., for cases where condition variables are used to synchronize parallel real-time computations, and possibly in presence of global multi-processor scheduling disciplines. Valuable starting points for such a kind of analysis might be the ones appeared in [44, 45].

9. ACKNOWLEDGEMENTS

This work was partially supported by the RETINA Eurostars Project E10171.

DOI: 10.1002/spe

References

1. Yan Y, Cosgrove S, Anand V, Kulkarni A, Konduri SH, Ko SY, Ziarek L. Rtdroid: A design for real-time android. *IEEE Transactions on Mobile Computing* Oct 2016; **15**(10):2564–2584, doi:10.1109/TMC.2015.2499187.
2. Kalkov I, Gurghian A, Kowalewski S. Priority inheritance during remote procedure calls in real-time android using extended binder framework. *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '15, ACM: New York, NY, USA, 2015; 5:1–5:10, doi:10.1145/2822304.2822311. URL `http://doi.acm.org/10.1145/2822304.2822311`.
3. The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition* 2004.
4. Cucinotta T. Priority inheritance on condition variables. *Proc. of the 9th International Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2013)*, Paris, France, 2013.
5. Sebesta RW. *Concepts of programming languages*. 10 edn., Addison Wesley, 2012.
6. Sha L, Rajkumar R, Lehoczky J. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on* sep 1990; **39**(9):1175 –1185, doi:10.1109/12.57058.
7. Abeni L, Manica N. Analysis of client/server interactions in a reservation-based system. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, ACM: New York, NY, USA, 2013; 1603–1609, doi:10.1145/2480362.2480662.
8. Corbet J. CFS group scheduling. http://lwn.net/ July 2007.
9. Hart D, Guniguntalay D. Requeue-pi: Making glibc condvars pi-aware. *Proceedings of the Eleventh Real-Time Linux Workshop*, 2009; 215–227.
10. Zijlstra P. Scheduling nightmares. `https://wiki.linuxfoundation.org/realtime/events/rt-summit2016/scheduling-nightmares`.
11. Hart D, Riegel T. Pthread condvars: Posix compliance and the pi gap. `https://wiki.linuxfoundation.org/realtime/events/rt-summit2016/pthread-condvars`.
12. Sha L, *et al.*. Real time scheduling theory: A historical perspective. *Real-Time Syst.* Nov 2004; **28**(2-3):101–155, doi:10.1023/B:TIME.0000045315.61234.1e.
13. Buttazzo GC. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd edn., Springer Publishing Company, Incorporated, 2011.
14. Lampson BW, Redell DD. Experience with processes and monitors in mesa. *Commun. ACM* Feb 1980; **23**(2):105–117, doi:10.1145/358818.358824. URL `http://doi.acm.org/10.1145/358818.358824`.
15. Cornhilll D, Sha L, Lehoczky JP. Limitations of Ada for real-time scheduling. *Proc. of the first international workshop on Real-time Ada issues*, IRTAW '87, ACM: New York, 1987; 33–39, doi:10.1145/36821.36798.
16. Cornhill D, Sha L. Priority inversion in Ada. *Ada Lett.* Nov 1987; **VII**(7):30–32, doi:10.1145/36072.36073.
17. Locke CD, Goodenough JB. A practical application of the ceiling protocol in a real-time system. *Proceedings of the second international workshop on Real-time Ada issues*, IRTAW '88, ACM: NY, 1988; 35–38, doi:10.1145/58612.59373.
18. Sha L, Rajkumar R, Lehoczky J. A priority driven approach to real-time concurrency control. *Technical Report*, CMU July 1988.
19. Rajkumar R. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the International Conference on Distributed Computing Systems*, 1990; 116–123.
20. Chen CM, Tripathi SK. Multiprocessor priority ceiling based protocols. *Technical Report*, College Park, MD, USA 1994.
21. Chen MI, Lin KJ. Dynamic priority ceilings: a concurrency control protocol for rt systems. *Real-Time Systems* Oct 1990; **2**(4):325–346, doi:10.1007/BF01995676.
22. Baker TP. Stack-based scheduling for realtime processes. *Real-Time Syst.* Apr 1991; **3**(1):67–99, doi:10.1007/BF00365393.
23. Lakshmanan K, Niz Dd, Rajkumar R. Coordinated task scheduling, allocation and synchronization on multiprocessors. *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, IEEE Computer Society: Washington, DC, USA, 2009; 469–478, doi:10.1109/RTSS.2009.51.
24. Schmidt D, Mungee S, Flores-Gaitan S, Gokhale A. Alleviating Priority Inversion and Non-Determinism in Real-Time CORBA ORB Core Architectures. *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, RTAS '98, IEEE Computer Society: Washington, DC, USA, 1998; 92–.
25. Abeni L, Buttazzo G. Integrating multimedia applications in hard real-time systems. *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998; 4–13.
26. Lamastra G, Lipari G, Abeni L. A bandwidth inheritance algorithm for real-time task synchronization in open systems. *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, 2001; 151 – 160, doi:10.1109/REAL.2001.990606.
27. Faggioli D, Lipari G, Cucinotta T. An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the linux kernel. *Proceedings of the* 4^{th} *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008)*, Prague, Czech Republic, 2008.
28. Faggioli D, Lipari G, Cucinotta T. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems* 2012; **48**:789–825. 10.1007/s11241-012-9162-0.
29. Block A, Leontyev H, Brandenburg B, Anderson J. A flexible real-time locking protocol for multiprocessors. *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, 2007; 47 –56, doi:10.1109/RTCSA.2007.8.
30. Guan N, Ekberg P, Stigge M, Yi W. Resource sharing protocols for real-time task graph systems. *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, Porto, Portugal, 2011.
31. Brandenburg BB, Anderson JH. Optimality results for multiprocessor real-time locking. *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, IEEE Computer Society, 2010; 49–60.

32. Behnam M, Shin I, Nolte T, Nolin M. Sirap: a synchronization protocol for hierarchical resource sharing real-time open systems. *Proceedings of the 7th ACM and IEEE international conference on Embedded software*, 2007.
33. Davis RI, Burns A. Resource sharing in hierarchical fixed priority pre-emptive systems. *Proceedings of the IEEE Real-time Systems Symposium*, 2006.
34. Easwaran A, Andersson B. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. *Proceedings of IEEE Real-Time Systems Symposium*, 2009.
35. Macariu G. Limited blocking resource sharing for global multiprocessor scheduling. *Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, Porto, Portugal, 2011.
36. van den Heuvel MM, Bril RJ, Lukkien JJ. Dependable Resource Sharing for Compositional Real-Time Systems. *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, 2011; 153–163, doi:10.1109/RTCSA.2011.29.
37. Yang M, Wieder A, Brandenburg BB. Global real-time semaphore protocols: A survey, unified analysis, and comparison. *2015 IEEE Real-Time Systems Symposium*, 2015; 1–12, doi:10.1109/RTSS.2015.8.
38. Xi S, Li C, Lu C, Gill C. Limitations and solutions for real-time local inter-domain communication in xen. *Technical Report* Oct 2012.
39. Steinberg U, Wolter J, Hartig H. Fast component interaction for real-time systems. *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, 2005; 89–97, doi:10.1109/ECRTS.2005.16.
40. Dragojević A, *et al.*. Why STM can be more than a research toy. *Commun. ACM* Apr 2011; **54**(4):70–77, doi: 10.1145/1924421.1924440.
41. Cucinotta T, Mancina A, Anastasi GF, Lipari G, Mangeruca L, Checcozzo R, Rusin'a F. A real-time service-oriented architecture for industrial automation. *IEEE Transactions on Industrial Informatics* August 2009; **5**(3).
42. Faggioli D, Lipari G, Cucinotta T. The multiprocessor bandwidth inheritance protocol. *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, 2010; 90–99.
43. Lelli J, Scordino C, Abeni L, Faggioli D. Deadline scheduling in the linux kernel. *Software: Practice and Experience* 2016; **46**(6):821–839, doi:10.1002/spe.2335. URL `http://dx.doi.org/10.1002/spe.2335`, spe.2335.
44. Chwa HS, Lee J, Phan KM, Easwaran A, Shin I. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013; 25–34, doi:10.1109/ECRTS.2013.14.
45. Li J, Agrawal K, Lu C, Gill C. Analysis of global edf for parallel tasks. *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013; 3–13, doi:10.1109/ECRTS.2013.12.